

Robust and Effective Malware Detection through Quantitative Data Flow Graph Metrics

Tobias Wüchner, Martín Ochoa, and Alexander Pretschner

Technische Universität München, Germany

Abstract. We present a novel malware detection approach based on metrics over quantitative data flow graphs. Quantitative data flow graphs (QDFGs) model process behavior by interpreting issued system calls as aggregations of quantifiable data flows. Due to the high abstraction level we consider QDFG metric based detection more robust against typical behavior obfuscation like bogus call injection or call reordering than other common behavioral models that base on raw system calls. We support this claim with experiments on obfuscated malware logs and demonstrate the superior obfuscation robustness in comparison to detection using n-grams. Our evaluations on a large and diverse data set consisting of about 7000 malware and 500 goodware samples show an average detection rate of 98.01% and a false positive rate of 0.48%. Moreover, we show that our approach is able to detect new malware (i.e. samples from malware families not included in the training set) and that the consideration of quantities in itself significantly improves detection precision.

1 Introduction

Despite the increasing availability and deployment of intrusion detection systems and anti-virus engines, malicious software (malware) remains a severe threat. One reason is the steadily increasing sophistication of modern malware. Most new malware families found in the wild employ some kind of functionality to avoid or harden detection by traditional security measures. Examples range from rather simplistic attempts to disable known security software upon infection; over polymorphism and metamorphism techniques to alter and obfuscate the executable binaries of malware in order to harden detection by signature-based approaches; up to more sophisticated behavioral obfuscation techniques, such as mimicry attacks, that aim at altering the runtime behavior to trick behavior-based detection approaches [35].

One challenge of malware detection research is thus the new threat of stealthy and obfuscated malware; and how to counteract their attempts to avoid detection and remain “below the radar”. We contribute towards this goal with a novel behavior-based malware detection methodology which we show to be less prone to circumvention by obfuscation mechanisms. The idea is to discriminate malicious from benign processes by analyzing their behavior *in terms of induced quantitative data flows between system resources*. We interpret the execution of system calls, e.g. a process calling the Windows API *ReadFile* function to read

data from a file, as causing a quantifiable flow of data from one system entity, in this example a file, to another entity, in this example a process.

We aggregate the set of data flow events in a system within a specific time interval into so-called quantitative data flow graphs (QDFGs). These represent the interaction between all system entities within this time frame. QDFGs are abstractions of a system’s behavior in terms of data flows, and thus can be used for behavior-based malware detection. On the basis of this model our approach aims at identifying QDFG nodes that refer to potentially malicious processes. To do so, we use metrics inspired by research done in the area of social network analysis, to profile typical data flow behavior of benign and malicious processes, and then use these profiles to train a machine learning classifier.

In contrast to related work on graph-based malware detection [32,9,13,12], we do not rely on fixed detection patterns and expensive subgraph isomorphism checks. Instead, we perform approximate similarity comparison of unknown process behavior with a more flexible metric-based quantitative data flow model. By this, in contrast to isomorphism-based approaches that are challenged if malware does not exactly match defined patterns or models, we are able to detect unknown or obfuscated malware. In contrast to recently published metric-based approaches [16,22] we incorporate quantitative data flow aspects into our model which we show to provide better detection precision and superior obfuscation resilience, as well as novel features (which we call *local*).

Contributions: **a)** To the best of our knowledge, we are the first to combine *quantitative* data flow tracking with machine learning for checking for behavioral similarity of processes in the context of malware detection. **b)** Our experiments demonstrate the utility of *quantitative* data flow aspects for detection precision. In particular we show that the consideration of quantities in data flow graphs can effectively halve false positive and false negative rates. **d)** Our evaluations indicate that our approach is more robust against common types of behavioral obfuscation than approaches that build on raw system calls such as n-gram based approaches and **e)** We show that we are able to detect samples from unknown malware families with good accuracy.

Organization: We recap an abstract QDFG model from the literature in §2. We present graph metrics and their semantic relevance in terms of malware detection in §3.1; describe the training phase in §3.2; and discuss the detection procedure in §3.3. We evaluate effectiveness, obfuscation robustness, and efficiency in §4. We put our approach in context in §5. We conclude with a discussion of capabilities, limitations, and future work in §6.

2 Preliminaries

We first recap some preliminaries. For the subsequent sections we assume a basic understanding of the Windows NT operating system architecture.

2.1 Quantitative Data Flow Model

We study the identification of potentially malicious processes in a system by analyzing quantitative data flow graphs (QDFGs) that represent a system’s data

flow activities within a certain period of time. To this end, we use a slightly simplified generic quantitative data flow graph model from the literature [32]. This model uses QDFGs to capture all aggregated and quantified data flows between interesting entities in a system, such as processes, files, or sockets. These are represented by nodes (\overline{N}). Labeled directed edges (\overline{E}) between two nodes intuitively reflect that there has been a transfer of a certain amount of data.

A QDFG is a graph in the set $\mathcal{G} = \overline{N} \times \overline{E} \times \overline{A} \times ((\overline{N} \cup \overline{E}) \times \overline{A} \rightarrow \text{Value}^{\overline{A}})$, where \overline{N} denotes the set of all possible nodes, $\overline{E} \subseteq \overline{N} \times \overline{N}$ the set of possible edges between two nodes, and a set of labeling functions $((\overline{N} \cup \overline{E}) \times \overline{A}) \rightarrow \text{Value}^{\overline{A}}$ assign defined values from the set $\text{Value}^{\overline{A}}$ to an attribute $a \in \overline{A}$ of a node or an edge. These labeling functions are needed to annotate nodes and edges with additional information such as amount of transferred data ($size \in \mathbb{N}$) or corresponding set of time stamps ($time \in 2^{\mathbb{N}}$).

QDFGs are incrementally built on the basis of data flow relevant system events, e.g. functions to read data from a file or to write data to a socket. These events are modeled as a set \mathcal{E} . In an actual system, they are intercepted by runtime monitors which interpret the data flow semantics and perform corresponding graph updates such as the creation or modification of nodes or edges. One QDFG $G = (N, E, A, \lambda) \in \mathcal{G}$ describes all data flow activities of a system that happened during a certain time interval. The labeling function λ maps attributes of an edge or a node to their assigned values.

Events $(src, dst, size, t, \lambda) \in \mathcal{E}$ represent transfers of $size \in \mathbb{N}$ units of data from a node $src \in \overline{N}$ to a node $dst \in \overline{N}$ with a timestamp $t \in \mathbb{N}$ and a labeling function λ for additional information on the corresponding data flow. To ease presentation, we will not always cleanly distinguish between an event and its corresponding edge. We are not interested in exactly which event causes which amount of data flow between two system entities. We thus simplify our model by aggregating semantically related data flows between pairs of nodes through summation of the $size$ attribute of the respective edges rather than creating one distinct edge per event.

Before formally defining the corresponding graph update function, triggered by the execution of a data flow related event, we first need to introduce some auxiliary notations and syntactic sugar: For $(x, a) \in (N \cup E) \times \overline{A}$, we define

$$\lambda[(x, a) \leftarrow v] = \lambda' \text{ with } \lambda'(y) = \begin{cases} v & \text{if } y \in \text{dom}(\lambda) \\ \lambda(y) & \text{otherwise} \end{cases}.$$

We furthermore introduce some syntactic sugar for updating labeling functions: $\lambda[(x_1, a_1) \leftarrow v_1; \dots; (x_k, a_k) \leftarrow v_k] = (\dots(\lambda[(x_1, a_1) \leftarrow v_1]) \dots)[(x_k, a_k) \leftarrow v_k]$.

Correspondingly we denote the composition of two labeling functions by:

$$\lambda_1 \circ \lambda_2 = \lambda_1[(x_1, a_1) \leftarrow v_1; \dots; (x_k, a_k) \leftarrow v_n]$$

where $v_i = \lambda_2(x_i, a_i)$ and $(x_i, a_i) \in \text{dom}(\lambda_2)$. Finally, the QDFG update function $update : \mathcal{G} \times \mathcal{E} \rightarrow \mathcal{G}$ is formally defined in Figure 1.

For later definitions of node features, we need to introduce auxiliary functions. Function $pre : \overline{N} \times \mathcal{G} \rightarrow 2^{\overline{N}}$ computes all immediate predecessor nodes of

$$\begin{aligned}
& \text{update}(G, (src, dst, s, t, \lambda')) = \\
& \left\{ \begin{array}{l} \left(\begin{array}{l} N, \\ E, \\ A \cup \text{dom}(\lambda'), \\ \lambda \left[\begin{array}{l} (e, size) \leftarrow \lambda(e, size) + s; \\ (e, time) \leftarrow (\lambda(e, time) \cup \{t\}) \end{array} \right] \circ \lambda' \end{array} \right) \quad \text{if } e \in E \\ \\ \left(\begin{array}{l} N \cup \{src, dst\}, \\ E \cup \{e\}, \\ A \cup \text{dom}(\lambda'), \\ \lambda \left[\begin{array}{l} (e, size) \leftarrow s; \\ (e, time) \leftarrow \{t\} \end{array} \right] \circ \lambda' \end{array} \right) \quad \text{otherwise} \end{array} \right. \\
& \text{where } e = (src, dst) \text{ and } G = (N, E, A, \lambda)
\end{aligned}$$

Fig. 1: Graph update function

a node of the graph. Functions $in, out : \bar{N} \times \mathcal{G} \rightarrow 2^{\bar{E}}$ compute the set of incoming and outgoing edges of a node.

2.2 Windows Instantiation

To instantiate the abstract QDFG model for real-world malware detection, we need to map it to resources and events in actual execution environments. In this paper, the execution environment is that of typical Windows operating systems.

We identified a set of system resources that are relevant for malware data flow behavior: *Processes* interact with all other relevant system entities in a way that they are either sources or sinks of flows from or to other *Registry*, *Socket* or *Process* nodes. To type these nodes we introduce a special $type \in \bar{A}$ attribute: Process nodes have $type P$, File nodes F , Socket nodes S , URL nodes U , and Registry nodes R .

In addition to *entities*, we also need to map all data flow relevant *events*. These are all Windows API functions that lead to a flow of data between the above system entities. This includes functions to interact with resources from the file system like *ReadFile* or *WriteFile* to functions to send or receive data to or from a socket like the Winsock *recv* and *send* functions. To give an intuition how the data flow semantics of such functions is formally modeled, we present two sample function definitions:

- **ReadFile** Using this function a process reads a specified amount of bytes from a file to its memory. *Relevant Parameters*: Calling Process (P_C), Source File (F_S), ToReadBytes (S_R). *Mapping*: $(F_S, P_C, S_R, t, \lambda(F_S, size) := \lambda(F_S, size) + S_R) \in \mathcal{E}$.
- **WriteFile** Using this function a process can write a specific number of bytes to a file. *Relevant Parameters*: Calling Process (P_C), Destination File (F_D), ToWriteBytes (S_W). *Mapping*: $(P_C, F_D, S_W, t, \lambda(F_D, size) := \lambda(F_D, size) + S_W) \in \mathcal{E}$.

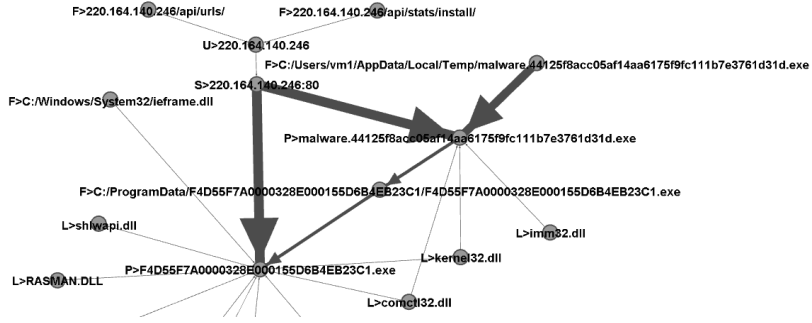


Fig. 2: Excerpt of QDFG for a system infected with Cleaman.

For brevity’s sake we only presented two functions here to demonstrate the general procedure of mapping concrete system events to abstract events of the QDFG model. A more comprehensive list can be found for instance in [32].

To motivate the utility of this model for malware detection, Figure 2 exemplarily visualizes a typical malware QDFG, built by applying the previously mentioned model on the intercepted activities of an executed Cleaman trojan. The size of the edges in the graph visualization represents the relative amount of transferred data with respect to all other edges of that graph. The type of the nodes is denoted by the first letter of the node label. By focusing on the part of the graph with the highest amount of transferred data one can easily spot the core malign activities of the analyzed malware, i.e. self-replication, or download and execution of additional malicious payload from a remote server.

3 Approach

Our core idea is to learn statistical profiles for benign and malicious nodes in QDFGs that represent known infected and non-infected systems. We later use these profiles for matching feature sets of unknown processes against them.

The overall architecture is depicted in Figure 3. Dashed lines mark components and interactions that are only used in the training phase. Dotted lines refer to the ones only relevant for detection.

3.1 Features

Like others [16,22] we see a strong analogy between social networks and (Q)DFGs and hence use graph characteristics inspired from social network analysis [25,5]. Nodes in a social network typically represent communicating entities, and the edges between them their interaction in form of exchanged messages or friendship relations. Analogously, nodes in our graphs represent system entities and the edges between them their interaction in form of data flows.

The following features were selected using both an inductive and a deductive approach. The inductive selection was done based on a preliminary analysis

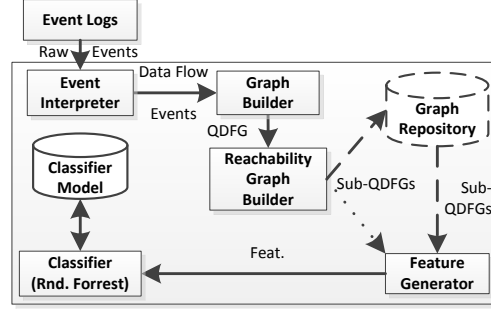


Fig. 3: Architecture

of graphs of a small set of malware-infected systems where we applied several standard metrics from statistics and graph theory and tried to correlate malware activities with the applied metrics. The deductive selection was performed through an analysis of standard graph metrics. For each analyzed feature we tried to correlate its intuition with typical malware behavior or properties. For instance, malware that tries to infect other processes or files results in a high connectivity of the corresponding node to certain types of other nodes.

Features are functions $\phi \in \Phi$ that map a QDFG node to a real number: $\phi : \overline{N} \times \mathcal{G} \rightarrow \mathbb{R}$. We enrich the QDFG model to store the value of features as attributes of nodes n in graph G , so $\lambda(n, \phi) = \phi(n, G)$. Additionally, we distinguish between two basic types of graph features, *local features* (Φ_l) and *global features* (Φ_g). Local features have a single-hop scope. This means that they only capture the relationship of a node with its direct neighbors. Global features in contrast have a multi-hop scope and represent relationships of one node with all other nodes of a graph.

As opposed to recently published metric-based approaches [16,22], that also exploit graph-theoretical properties to derive discriminating features, we take into account quantitative data flow aspects by exploiting the additional information given by the weighted edges of QDFGs.

Local Features (Φ_l) To define the features, we need some auxiliary notation. Function $d_\psi : (\overline{N} \times \overline{N} \times \mathcal{G}) \rightarrow \mathbb{R}$ with ψ returns the shortest path between two nodes in a graph, where $\psi : \overline{E} \rightarrow \mathbb{R}$ with $\psi(e) = \lambda(e, size)$ defines the edge distance, or cost, i.e. the amount of data transferred via this edge.

1. **Entropy** $\phi^1 \in \Phi_l$ computes the normalized entropy of the distribution of edge feature values such as size, event count, or sensitivity of all outgoing edges of a process node $n \in N$. The entropy captures the uniformity of the distribution of percental flows, number of contributing events, or relative sensitivity of all outgoing edges of a node n .

Rationale: Viruses like Parite infect other executable binaries or processes by injecting or appending their own binary image. The respective subgraphs

tend to have a comparably uniform distribution of specific features of outgoing edges, because the majority of triggered events by that malware are targeted at the infection with roughly the same size of the events as consequence of them relating to reading or writing the same binary image.

Computation: Let $\vec{s} = (s_1, \dots, s_k)$ and $e_i \in out(n)$ in $s_i = \frac{\psi(e_i)}{\sum_{e' \in out(n)} \psi(e')}$.

Then we define: $\phi^1(n, G) := NE(\vec{s})$ where $NE(\vec{s}) := \frac{-\sum_{i=1}^k s_i * \log(s_i)}{\log(k)}$.

2. **Variance** $\phi^2 \in \Phi_l$ expresses the statistical population variance of the distribution of a certain edge feature for all outgoing edges of a node $n \in N$. A low statistical variance indicates that most of the elements of the distribution elements are close to the statistical mean, whereas a high variance indicates a spread of elements. Due to its similar focus on uniformity of underlying input distributions, the variance feature is closely correlated with the entropy feature. First evaluations indicated, however, that the entropy metric performed comparably badly if a node has only a few outgoing edges, but better for larger sets of outgoing edges. The variance metric seemed to exhibit exactly the inverse characteristics.

Rationale: The motivation is similar to that for entropy: malware often exhibits outgoing edge distribution characteristics different from benign ones.

Computation:

$$\phi^2(n, G) := \frac{\sum_{e \in out(n)} \left(\psi(e) - \frac{1}{|out(n)|} \sum_{e' \in out(n)} \psi(e') \right)}{|out(n)|}$$

3. **Flow Proportion** $\phi_t^3 \in \Phi_l$ captures the proportion of a certain type of outgoing data flows of a node $n \in N$ w.r.t. all outgoing flows of that node. The type of a flow is determined by the target node's type of the outgoing edge. We define different variants of the proportion feature that consider different edge attributes.

Rationale: Malware processes often exhibit different flow proportion characteristics than goodware. Examples include ransomware or virus processes that have an irregularly high percentage of outgoing edges that point to file nodes, as they either encrypt several sensitive files, or infect all executable binary files on the hard disk.

Computation: Let $t \in \{Process, Registry, File, Socket\}$.

$$\phi_t^3(n, G) := \frac{\sum_{e=(src,dst) \in out(n), \lambda(dst,type)=t} \psi(e)}{\sum_{e \in out(n)} \psi(e)}$$

Global Features (Φ_g) Global features represent the relation between one node and—possibly all—other nodes of a graph. In contrast to local features, capture the importance of one node within the overall graph. Note that a crucial feature

of global features is the fact that the weight of edges (given by the size of data flows between them) is considered when computing the shortest path between nodes (given by the function $\psi(e)$).

1. **Closeness Centrality** $\phi^4 \in \Phi_g$ for a node $n \in N$ represents the inverse of that node's average distance to all other nodes of the same graph. A high closeness centrality indicates that the respective node is closely connected to all other graph nodes [25].

Rationale: High connectivity with other nodes indicates a node manipulating or infecting other system resources like processes or executable binaries. Such behavior is typical for viruses like Parite that replicate by infecting other processes and binaries. This leads to a close connectivity of the corresponding malware process node with other process and binary file nodes.

Computation:

$$\phi^4(n, G) := \frac{|N| - 1}{\sum_{n' \in N \setminus \{n\}} d_\psi(n, n', G)}$$

2. **Betweenness Centrality** $\phi^5 \in \Phi_g$ of a node $n \in N$ represents the relative portion of all shortest paths between all possible pairs of nodes of a graph that pass through that specific node n . A high betweenness centrality means that one specific node is part of a multitude of “communications” between nodes [25].

Rationale: This metric captures how often a process is part of a multi-step interaction or data flow between other system resources. This is useful to identify malware aiming at man-in-the-middle attacks to e.g. intercept the communication of a benign process with a socket, or to manipulate the information that a benign process reads into memory, to e.g. infect that process with malicious code at runtime.

Computation: The function $sp(x, y, G)$ returns the number of shortest paths between the nodes x and y in a graph G ; $sp_z(x, y, G)$ the ones that pass through node z .

$$\phi^5(n, G) := \sum_{n', n'' \in N: n' \neq n''} \frac{sp_n(n', n'', G)}{sp(n', n'', G)}$$

3.2 Training and Model Building Phase

We can now establish statistical profiles for the discrimination between benign and potentially malicious process nodes in a graph. A concrete instantiation of this training procedure with real-world data will be discussed in §4.2. The training procedure consists of four activities: i) event log generation; ii) graph generation; iii) feature extraction; iv) classifier training.

Event Log Generation Using a user mode Windows API monitor from the literature [33], we log a defined amount of calls of processes to the Windows API to capture the activity and interaction of all processes within a system for a certain period of time. The monitor intercepts process calls to the Windows API and stores data flow relevant information like event name, parameter values, and name of the issuing process along with additional context information like a time-stamp to an event log.

Graph Generation We then extract the data flow related information from the event logs. This is done by an *event data flow interpreter* component that maps raw events to the semantic model discussed in §2.2. The *graph builder* then generates one QDFG per event log as described in §2.1. In order to reduce noise, instead of storing the complete QDFG for training, we generate so-called reachability graphs for all process nodes in the base QDFGs. Such reachability graphs contain all nodes and edges that are directly or indirectly connected to the starting node. By this means we ensure, that the training graphs only contain activities that are actually triggered by a certain process or of processes that it directly or indirectly influenced, ignoring all activities that are conducted by non-related processes.

Feature Extraction The *feature extractor* computes all graph features from §3.1 for all process nodes of the graphs in the training graph repository. This yields a set of feature values for different benign and malicious process. Recall that we labeled the known malicious process and thus also the corresponding graph nodes. We are hence able to label the resulting process node feature sets as belonging to a known malicious/benign process, which is a necessary precondition for later using a supervised machine learning algorithm.

Classifier Training After the feature extraction phase, we feed the obtained features into a machine learning algorithm for training.

For this we construct a feature vector for each process node of the training set, with the elements of the vector being the considered QDFG metrics (see §3.1), together with one label element representing the known classification (benign or malign) of the respective process. Each feature vector is thus of size $|\Phi| + 1$.

Note that we only compute feature vectors for process nodes as we are solely interested in determining whether a specific process that originated from a executed binary is malicious or not; we thus do not classify a binary itself, but its runtime representation, i.e. the respective process or its children.

Considering the high number and diversity of the value space of the selected training features we need a machine learning algorithm that is robust towards training set diversity and scales well with respect to the number of training features. Initial attempts to use simple classifiers like naive Bayes yielded poor performance in terms of detection precision. We hence explored more complex algorithms like support vector machines and meta-learners. Particularly good

results were achieved with the Random Forest (RF) algorithm. RF is a meta- or ensemble-learner, which means that it uses several distinct, potentially imprecise, classification models and merges their decisions to form a more precise combined decision. RF constructs many individual decision trees, called decision forest, based on random selection of limited feature subsets of the feature space.

3.3 Detection Phase

We now have a classifier that can predict the class (malicious or benign) of an unknown process node based on its characteristic local and global graph features. In a nutshell for the detection phase we thus only need to build a graph of a potentially infected system at runtime based on captured events, compute the characteristic features for each process node in the graph, and match the resulting feature set against the classifier.

Like for the training phase, we intercept relevant system events at runtime, interpret them in terms of their data flow semantics, and then build the corresponding (reachability) QDFGs for each process. We then compute the characteristic feature sets for the process nodes of the generated reachability graphs and match them against the classifier, using the classification model that was generated as result of the training phase.

Consequently, all process nodes of these reachability graph are classified into benign or potentially malicious ones.

4 Evaluation

We implemented the detection framework and captured activities of a representative and diverse set of known benign and malicious software to assess the effectiveness and efficiency of our approach.

4.1 Prototype

Our prototype is a distributed system as shown in Figure 3. We used and extended a user mode Windows API runtime monitor [33] to intercept system activities relevant in terms of data flows of all processes running within the evaluation system. For the training phase we used the Random Forest implementation of the Weka machine learning framework [15], configured to build a forest of 10 distinct decision trees using the a random feature subsets.

To be able to analyze a large body of malware samples it was necessary to automate the different analysis steps. For this purpose we customized the open-source malware sandbox framework Cuckoo ¹ by replacing its function call hooking module with our hooking module. Each Cuckoo sandbox VirtualBox instance was running a clean installation of Windows 7 SP1 and assigned two 2,4GHz cores and 2GByte of RAM. The generation of the QDFGs, computation of the corresponding graph features, and the classification of process samples performed on a 2,8GHz quadcore i7 system with 8GByte of RAM.

¹ <http://www.cuckoosandbox.org/>

4.2 Effectiveness

As data source for our experiments we used 6994 different known malicious programs and 513 different known benign applications.

The malicious program samples were taken from a subset of the Malicia malware data set, i.e. all samples that were executable in the considered evaluation environment, that comprises of real-world malware samples from more than 500 drive-by download servers [24]. The respective malware set consists of samples from 12 malware families, including families like zeus, spyeye, and ramnit.

The goodwill set was composed of a selection of popular applications from <http://download.com> and a wide range of standard windows programs, including popular email programs like ThunderBird, browsers like FireFox, video and graphics tools like Gimp, or VLC Player, and security software like Avast.

We generated about 7500 event logs and converted them into QDFGs, each capturing activities of the sandbox machines for a time interval of 5 minutes.

With this data and the procedure explained in the previous section we obtained a total of 8648 (i.e. 1654 goodwill and 6994 malware) QDFG feature sets. The reason for the set of goodwill features being bigger than the set of executed goodwill samples is that for each execution of a goodwill sample we did not only capture the behavior of the goodwill sample itself, but also the interaction with all simultaneously running standard Windows processes.

To evaluate the detection performance of our approach on the obtained feature set we first performed ten times a 10-fold cross validation test. For this tests we split the entire feature set into two parts, using 90% of the set for training and the remaining 10% for testing. The sets were randomly generated and the splitting repeated 10 times for each test to limit bias from specific set compositions. For each run we built a classification model on basis of the training data and used it for classifying the remaining test set. To avoid training bias due to unbalanced feature sets we in addition applied a SMOTE oversampling [8] on the training sets to approximately balance the distribution of malware and goodwill samples. To express the effectiveness, we computed the following quality metrics. True positives (TP) refer to malware samples (MW) that have been correctly classified as malicious, true negatives (TN) to goodwill samples (GW) that were correctly classified as benign, false positives (FP) to goodwill samples incorrectly classified as malicious, and false negatives (FN) malware samples that were mistakenly labeled as benign:

$$\begin{aligned} \text{Detection Rate (DR)} &: \frac{TP}{MW} & \text{False Positive Rate (FPR)} &: \frac{FP}{GW} \\ \text{Precision} &: \frac{TP}{TP+FP} & \text{F-Measure} &: \frac{2*TP}{2*TP+FP+FN} \end{aligned}$$

Table 1 (a) depicts the average *effectiveness quality metrics* of the cross validation experiments. As we can see, our approach at average can correctly detect 98 % of the provided malware set with a low false positive rate of only about 0.5%. The low standard deviations furthermore indicates a good stability of the results.

| | a) Real | b) Fixed | c) Random |
|-------------------------------|---------------------------------|---------------------------------|---------------------------------|
| Avg. Det. Rate (Std. Dev.) | 98.01% ($\sigma = 0.51\%$) | 98.00% ($\sigma = 0.57\%$) | 95.23% ($\sigma = 0.90\%$) |
| Avg. FP Rate (Std. Dev.) | 0.48% ($\sigma = 0.34\%$) | 0.85% ($\sigma = 0.35\%$) | 1.08% ($\sigma = 0.41\%$) |
| Precision | 99.62% | 99.32% | 99.12% |
| F-Measure | 98.81% | 98.65% | 97.13% |

Table 1: Effectiveness Quality Metrics

Impact of Quantities To evaluate our hypothesis, that the consideration of quantities has a significant impact on the effectiveness of the classification, we performed two more tests. For the first test we replaced the real quantities associated to the edges of the QDFGs with a globally fixed value of 1. For the second test we performed the edge quantity replacement by associating varying random quantities to the edges. With this we effectively destroyed the inherent quantitative information of the QDFGs. For both experiments we again performed 10-fold cross validation tests to ensure stability of the results. Table 1 (b) and (c) depicts the average detection and false positive rates for both settings.

To calculate the relative impact of quantities on the detection effectiveness we divided the false positive and false negative rate (which is the dual of the detection rate) for the fixed and randomized quantities experiment by the respective rates of the experiment with the real quantities.

As we can see, fixing the quantities to a constant value increases the false positives by a factor of 1.8 ($\frac{.0085}{.0048}$). For the randomized quantities experiment we could observe an even bigger loss of effectiveness. Here the false positives increased by a factor of 2.3 ($\frac{.0108}{.0048}$), while also the false negatives increased by a factor of 2.4 ($\frac{1-.9523}{1-.9801}$) with respect to the experiments with the actual quantities.

These observations thus support the hypothesis about the utility of quantitative information for malware detection. To verify the statistical significance of these finding we performed a two-tailed t-test on the detection and false positive rates of the different experiments. The resulting p-values were all far below 0.01%, which indicates a high statistical significance of our observation.

Ability to Detect New Malware For evaluating our second hypothesis, i.e. that we are able to detect new malware, we performed an additional classification experiment. For each experiment run we split our data set into two parts. The first part, which we used for training, contained all goodware samples and the samples of all malware families except for one. The second set correspondingly contained all samples from the remaining family and was used as test set.

With this strategy we ensured that the training set did not contain any samples from the same family that was used for testing. In consequence the classifier could not gain any knowledge about the to be classified malware family.

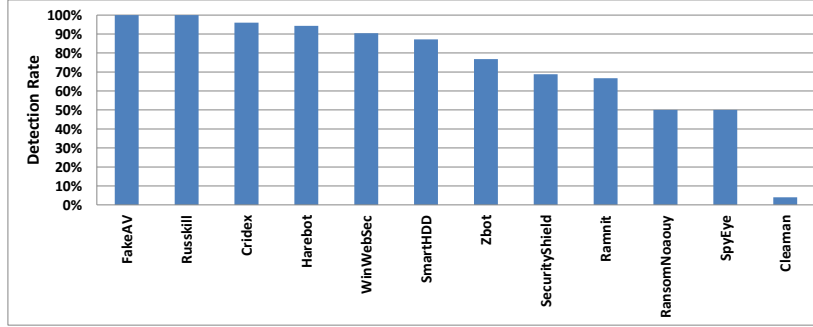


Fig. 4: Detectability of new malware

With this test procedure we simulated the real-world scenario that our approach faced a sample from a new malware type that was never seen before and thus could not be used for training the detection model.

Figure 4 depicts the detection rates that could be achieved for the different malware families. Each bar shows the percentage of all malware samples of a specific family that could be detected using a classifier that was trained on the samples from the remaining malware families.

As we can see, our approach in all cases was able to detect samples from unknown malware families. On average our approach was able to correctly identify 73.68% of the new malware samples; some malware families could even be classified with 100% correctness. These results for this experiment supports our hypothesis that our approach is capable of detecting new unknown malware and goodware. Although further investigation is needed to understand why, we speculate that this is the case because malware behavior is often not too different, even between distinct malware families.

Obfuscation Resilience Approaches that obfuscate the binary image of malware through build-time code encryption and run-time decryption or code diversification barely have any influence on the detectability through behavior-based detection approaches as such code transformations typically do not alter the externally observable program behavior. In consequence, our approach is likely to be widely robust to such used code obfuscation. This assumption is supported by the ability of our approach to detect variants of malware families that were obfuscated through code transformations. Our evaluations for instance show that we were able to detect 96 of 101 variants of the Harebot trojan from our data set, which is known to employ different forms of code obfuscation.

On the other hand, if malware e.g. non-deterministically executes bogus non-malicious activities or randomly alters between semantically equivalent system calls to achieve the same behavior, it can effectively trick common behavioral approaches that base on n-grams profiling and re-identification as consequence on the unpredictable diverse resulting n-grams as we will show in the following. The same holds for most call-graph based approaches as call-graphs can be easily obfuscated by altering or reordering system calls.

Our approach is by construction more robust against call reordering or substitution approaches. This is because reordering of system calls does not alter the corresponding QDFGs, and because semantic substitutions of system calls typically exhibit similar data flow properties that result in similar QDFG updates. Moreover, the injection of bogus calls can change QDFGs, in particular if new edges are created in consequence of e.g. previously untouched system entities being read or written to, or if certain operations are repeated such that the edge weights are altered.

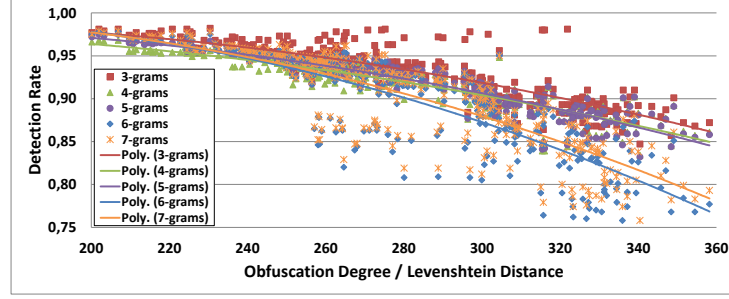
To evaluate the absolute effects of different types of behavioral obfuscation techniques on the effectiveness of our approach we thus set up a series of additional experiments.

First, we picked a set of 100 malware and 100 goodware samples as baseline for our experiments. To reason about the obfuscation resilience of our approach and related behavioral detection approaches like n-gram based ones we then step-wise applied behavioral obfuscation transformations on the call traces of these samples to achieve two typical types of behavior obfuscation, namely reordering of calls and injection of bogus calls. We did so by applying a behavior obfuscation tool [2] to the baseline malware set. This tool obfuscates commodity malware by randomly reordering its issued, or injecting new, system calls.

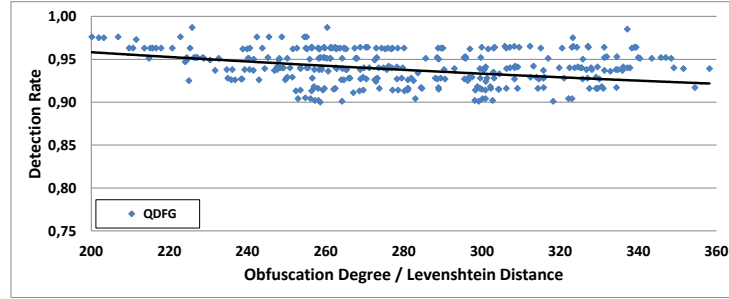
To investigate the effect of increasing degree of behavior obfuscation, we repeated these obfuscation steps 360 times with different configurations for call reordering and injection probabilities, as well as different upper bounds for to be reordered or injected calls. We represented the degree of obfuscation as Levenshtein Distance between the obfuscated and the non-obfuscated baseline call traces. More specifically, we computed the average number of insertions, deletions, and substitutions needed to transform the non-obfuscated call traces into the respectively obfuscated ones. To also get a relative comparison with other behavioral, raw system call based detection approaches, we conducted these experiments with measuring both, the detection effectiveness of our approach and the effectiveness of a typical behavioral detection approach based on n-grams of unordered system calls without arguments with varying n-gram sizes as e.g. discussed in [6].

As we can see in Figure 5, the ability of the n-gram based approach to correctly identify malicious processes significantly dropped with increasing obfuscation degree. In particular, higher-order n-grams seem to be more sensitive to behavioral obfuscation than lower-order ones. In contrast to n-gram based approaches, our approach seems to be considerably more robust and stable towards behavioral obfuscation. While the effectiveness of the n-gram approaches quickly dropped quadratically and lost prediction stability when obfuscation transformations were applied, the effectiveness of our approach at the same time remained rather stable and only slowly dropped linearly.

In sum, our evaluation indicates that we are rather robust with respect to realistic behavior obfuscation such as random bogus call injection or reordering, whereas we could show that common n-gram based approaches are considerably challenged by such obfuscation techniques.



(a) n-gram classifier



(b) QDFG metric classifier

Fig. 5: Obfuscation experiments

4.3 Efficiency

The generation of Random Forest classification models took between 55.21 and 75.38 seconds. The size of the generated models was between 16 and 19 MBytes. As the training and model generation phase is only conducted once, this overhead does not contribute to the overhead during the detection phases.

As we can see in Figure 6 the overall detection time seems to increase quadratically with respect to the graph size. The bottom-most part of the area stack refers to the time it took to generate a QDFG from a given event log, the area on top of that indicates the time needed to compute the local graph metrics, and the top-most area expresses the time spent for computing the global metrics. This is not surprising, as most graph algorithms such as the used centrality metrics have a theoretical complexity of $\mathcal{O}(n^2)$ to $\mathcal{O}(n^3)$, with n being the number of nodes in the graph [5]. On the other hand, the overhead for graph generation and computation of local features only grows linearly with respect to the graph size. The overhead induced by matching the generated graph features against the classification model was below the evaluation precision threshold of 1 ms and thus ignored for this analysis as it has no noticeable impact on the overall overhead. The size of the QDFGs used for our evaluation ranged from 47 to 330 edges resulting in an overall detection time between 24ms and 412ms. On aver-

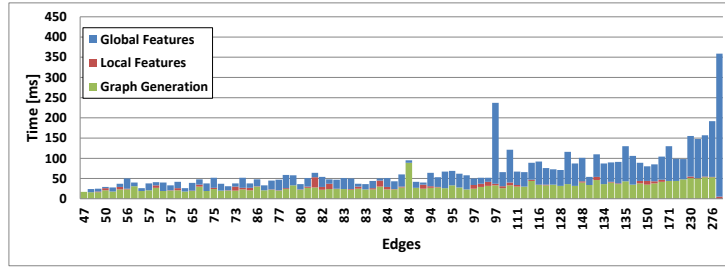


Fig. 6: Computation Time vs. Graph Complexity

age, generating a QDFG, computing its local and global features, and matching it against the trained classifier took 71.213ms.

4.4 Summary and threats to validity

In sum, we have shown our approach to be highly effective; to yield these results, among other things, on the basis of quantities; to be able to detect malware families for which it has not been trained; to be robust w.r.t. obfuscation – in particular more robust than n-grams; and to be efficient.

Naturally, these results need to be put in the context of our study. The evaluation results were obtained on the basis of a proof-of-concept prototype with event logs generated in a controlled lab environment. The obtained insights might not generalize to the application of our approach in real-world settings.

We tried to limit the risk of over-fitting the model to specific sub-sets of the training data by pro-actively diversifying the training set by selecting a wide and diverse range of popular malware and goodware samples for our training set. Furthermore, the fact that our cross-validation results show a very low standard deviation supports the assumption that we did not over-fit our models.

We executed and sampled the evaluation malware in a virtualized sandbox environment. It is known that modern malware often includes functionality to detect virtualization. There hence is a risk that we train our classification models on unrealistic malware behavior and thus are not able to detect their behavior in real-world scenarios. Even though we did not explicitly address this threat, for the future we plan to generate malware event data on realistic bare-metal environments and re-run the experiments to find out if this has any effect on the performance of our approach.

Because current malware still mainly focuses on avoiding detection by signature-based approaches, it rarely employs advanced behavior obfuscation techniques. Although we applied as much randomization as possible for our obfuscation it can of course not be excluded that the simulations do not adequately reflect real-world obfuscation techniques, or that adversaries might come up with more complex behavioral obfuscation operators.

5 Related Work

While behavior-based detection algorithms can still not be considered standard for commercial products, they have a long history in academia, especially in the intrusion detection domain. We do not perform a full literature survey on this topic here but focus on dynamic behavior-based malware detection.

A seminal idea of Forrest et al. [11] for behavior-based analysis was to profile benign and malign processes on the basis of characteristic system call sequences (n-grams). This approach was later refined and combined machine learning methods to improve classification effectiveness [21,14,19,23,31]. Similar ideas were also used to classify malware w.r.t. behavioral similarities [1,28,29].

Our system also intercepts and processes system calls. However, while the previously mentioned approaches directly use sequences of raw system calls, we use a *data flow abstraction* of these calls. This distinction is important as approaches that base on raw system call data are often significantly challenged by malware that use behavioral obfuscation techniques [4,30,35]. Such obfuscations for example target at breaking n-gram based approaches by reordering or randomizing system call sequences. As we base our analysis on a (quantitative) data flow abstractions of the system calls, which is independent of sequence order, our approach is, by construction, more robust to such obfuscation attempts.

Besides those detection approaches that use non-interpreted sequences of raw system calls, a separate line of research performs intermediate interpretation steps. A common approach is to extract semantic dependencies between different system calls of a process to form characteristic profiles for known goodware and malware. Popular examples represent system call dependency profiles in form of data or control flow graphs [18,26,13,20], or assign high-level semantics to known graphs [10,9,27]. (Sub-)graphs that pertain to known malicious behavior are then used to re-identify malicious behavior of system processes at runtime. Due to the used intermediate abstraction steps such approaches also appear more robust to behavioral obfuscation attempts. Unfortunately, they are also challenged in terms of identifying previously unknown malicious behavior for which detection profile graphs have not yet been extracted.

In particular, from the data flow perspective, close to ours is the work of Park et al. [26]. They construct Data Flow Graphs based on system calls, where entities are processes, child processes and files. Based on the DFGs of variants of a given malware family, they compute a common sub-graph called a *HotPath*. This process can be repeated for sets of variants of different malware families, and the resulting sub-graphs can be used to classify the DFG of a given process: it will be a variant of a known malware family if it contains a similar sub-graph or goodware otherwise. Different from our work, they do not consider quantities in their graph representation (which we have shown to be an important discriminating factor) and by construction their approach is tailored to recognize mutations of known malware families. They discuss the robustness of their approach against similar obfuscation techniques as the ones we consider, but opposed to our work their approach is challenged by the injection of arbitrary bogus system calls.

Other approaches [34,17] share similar drawbacks, as they depend on explicit definitions of malicious behavior. Our approach in contrast does not rely on fixed detection patterns. Due to the generic high-level nature of the used data flow graph features, it is more likely to also be able to detect new attacks and malicious behavior that deviate from the ones that were used for training. The use of statistical graph-based metrics for detection instead of fixed data flow patterns also differs from previous work on malware detection through quantitative data flow graphs [32]. The main difference to related work that leverage taint analysis for anomaly-based malware detection [3,7] is that we leverage quantitative data flow aspects without using comparably expensive taint tracking.

The recently published work of Jang et al. [16] relates to our work in that they also leverage graph metrics to discriminate malware from goodware. But, in contrast to our work, they base the computation of those metrics on system call dependency graphs, while our model is based on quantitative data flow graphs. As we could show, this abstraction increases the robustness towards behavioral obfuscation which gives us better resilience than approaches that directly base on raw system calls. Mao et al. [22] also leverage graph metrics on system entity dependency graphs for malware detection. Similarly, in contrast to us they do not incorporate any quantitative flow information for which we could show that it has an considerable impact on detection precision.

In sum, the main technical difference between our work and related contributions is that we leverage QDFG features rather than raw system calls or system entity dependency graphs. This makes our approach fast and robust against common types of behavioral obfuscations, and, due to the additional quantitative dimension, we achieve a good detection precision.

6 Discussion and Conclusion

We have presented a novel approach to perform graph metric based malware detection on the basis of quantitative data flow analysis. We intercept system calls issued by system processes, interpret them in terms of their data flow semantic and build quantitative data flow graphs. These are used to identify graph nodes that represent potentially malicious system processes.

To this extent, we compute sets of characteristic graph features, such as centrality metrics, for each process node in the graph to discriminate between benign and potentially malign nodes through a machine learning classifier. Using this classifier, trained on feature sets of known goodware and malware, we are able to discriminate unknown process samples.

It is difficult to objectively compare the effectiveness of different malware detection approaches presented in literature, due to varying evaluation baselines and used assessment procedures. However with respect to closely related dynamic malware detection approaches [34,18,13], our approach has similar or better detection effectiveness, while achieving a significantly better efficiency.

In contrast to previous QDFG-based work [32] and related rule-based approaches [34], our approach is able to detect novel malware samples that exhibit

unknown behavior with better detection effectiveness and efficiency. In comparison to related metric-based approaches [16,22] we could show that the quantitative aspect significantly improves detection precision.

Moreover, we have shown that our approach is robust to certain classes of behavioral obfuscation: by construction the *order* of system calls is irrelevant, since they produce the same QDFGs, and more interestingly, random injection of system calls that potentially modify both the structure and the original quantities does not significantly alter the detection effectiveness either.

In conclusion, we showed the usefulness of quantitative data flows for malware detection and established a foundation for further research in the area of QDFG based malware detection models. We plan to perform further tests on the robustness of our approach, try to generalize our approach to non-sandbox settings, and to improve effectiveness through additional graph features.

References

1. M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated classification and analysis of internet malware. In *RAID*, 2007.
2. S. Banescu, T. Wüchner, M. Guggenmos, M. Ochoa, and A. Pretschner. FEEBO: An empirical evaluation framework for malware behavior obfuscation. *CoRR*, arXiv:1502.03245, 2015.
3. S. Bhatkar, A. Chaturvedi, and R. Sekar. Dataflow anomaly detection. In *S&P*, pages 15–pp. IEEE, 2006.
4. J.-M. Borello and L. Me. Code obfuscation techniques for metamorphic viruses. *J. in Computer Virology*, pages 211–220, 2008.
5. U. Brandes. A faster algorithm for betweenness centrality. *J. of Mathematical Sociology*, 25(2):163–177, 2001.
6. D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. A quantitative study of accuracy in system call-based malware detection. In *ISSTA*. ACM, 2012.
7. L. Cavallaro and R. Sekar. Taint-enhanced anomaly detection. *Information Systems Security*, pages 160–174, 2011.
8. N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: synthetic minority over-sampling technique. *J. of Artificial Intelligence Research*, 16(1):321–357, 2002.
9. M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *India Software Engineering Conference*, pages 5–14, 2008.
10. M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant. Semantics-Aware Malware Detection. *S&P’05*, pages 32–46, 2005.
11. S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for Unix processes. *S&P*, pages 120–128, 1996.
12. M. Fredrikson, M. Christodorescu, and S. Jha. Dynamic behavior matching: A complexity analysis and new approximation algorithms. *Automated Deduction-CADE-23*, pages 252–267, 2011.
13. M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *S&P*, pages 45–60. IEEE, 2010.

14. A. K. Ghosh, A. Schwartzbard, and M. Schatz. Learning program behavior profiles for intrusion detection. In *Workshop on Intrusion Detection and Network Monitoring*, pages 6–6, 1999.
15. M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.
16. J.-w. Jang, J. Woo, J. Yun, and H. K. Kim. Mal-netminer: malware classification based on social network analysis of call graph. In *WWW*, 2014.
17. E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer. Behavior-based spyware detection. In *USENIX*, 2006.
18. C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-y. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *USENIX*, pages 351–366, 2009.
19. A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. Accessminer: Using system-centric models for malware protection. In *CCS*, pages 399–412, 2010.
20. J. Lee, K. Jeong, and H. Lee. Detecting metamorphic malwares using code graphs. *SAC*, 2010.
21. W. Lee, S. J. Stolfo, and P. K. Chan. Learning patterns from unix process execution traces for intrusion detection. In *Workshop on AI Approaches to Fraud Detection and Risk Management*, pages 50–56, 1997.
22. W. Mao, Z. Cai, X. Guan, and D. Towsley. Centrality metrics of importance in access behaviors and malware detections. In *ACSAC*. ACM, 2014.
23. N. A. Milea and S. C. Khoo. Nort : Runtime anomaly-based monitoring of malicious behavior for windows. pages 115–130, 2012.
24. A. Nappa, M. Z. Rafique, and J. Caballero. Driving in the Cloud: An Analysis of Drive-by Download Operations and Abuse Reporting. In *DIMVA '13*, Berlin, Germany, July 2013.
25. K. Okamoto, W. Chen, and X.-Y. Li. Ranking of closeness centrality for large-scale social networks. In *Frontiers in Algorithmics*, pages 186–195. Springer, 2008.
26. Y. Park, D. S. Reeves, and M. Stamp. Deriving common malware behavior through graph clustering. *Computers & Security*, 2013.
27. M. Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. *ACM SIGPLAN Notices*, pages 1–12, 2007.
28. K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and classification of malware behavior. In *DIMVA*, pages 108–125. Springer, 2008.
29. K. Rieck, P. Trinius, C. Willems, and T. Holz. Automatic analysis of malware behavior using machine learning. *J. of Computer Security*, pages 639–668, 2011.
30. M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *NDSS*, 2008.
31. C. Wressnegger, G. Schwenk, D. Arp, and K. Rieck. A close look on n-grams in intrusion detection: anomaly detection vs. classification. In *Workshop on Artificial intelligence and security*, pages 67–76, 2013.
32. T. Wüchner, M. Ochoa, and A. Pretschner. Malware detection with quantitative data flow graphs. In *ASIACCS*, 2014.
33. T. Wüchner and A. Pretschner. Data loss prevention based on data-driven usage control. In *ISSRE*, pages 151–160, Nov 2012.
34. H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *CCS*, pages 116–127, 2007.
35. I. You and K. Yim. Malware obfuscation techniques: A brief survey. In *BWCCA*, pages 297–300, 2010.